Lecture 5

Stacks

Stacks and queues versus normal lists

- Normal lists may be static in size such as Array or dynamic in size such as ArrayList(List(object) or List(of T).
- There are some application where its more naturally to use a list with some restrictions on the operation that could be performed on it.
- In modeling customer services at a bank teller we need a list such that customer enter the list from one end and served from the other: Queue
- In implementing mathematical expression evaluation and function calls, we need a list such that the addition to, and the removal from, it occur at one end: Stack
- Using such restricted list make it more naturally and prevents errors

Stacks

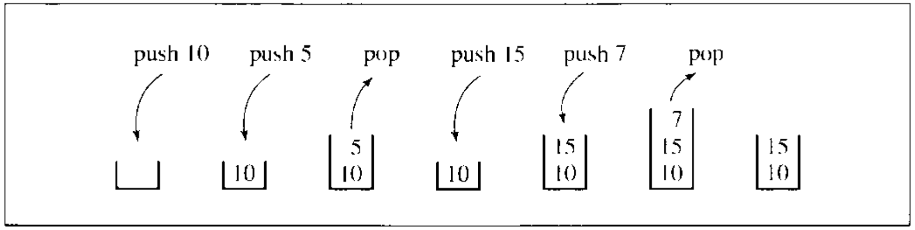
A stack is a Last In First Out(LIFO) data structure

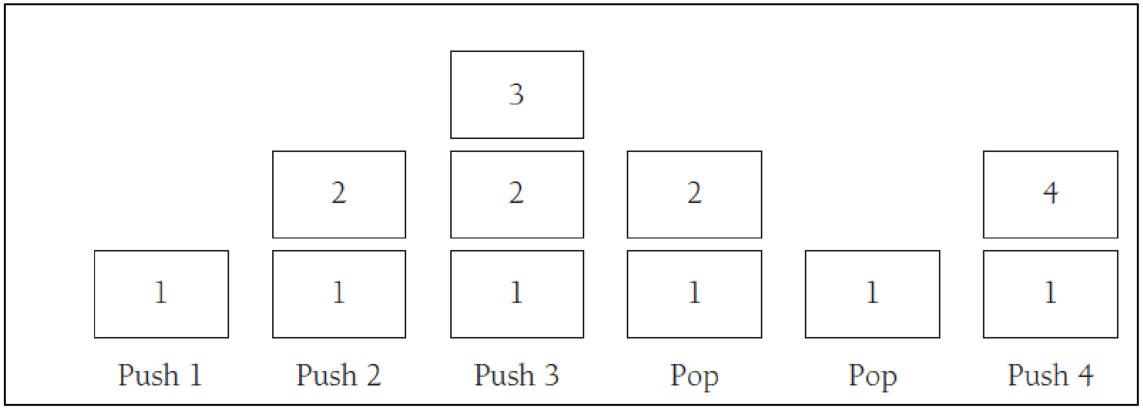
Principal operations

- Putting a data item in the stack is called “push”
- Getting out a data item of the stack is called “Pop”. This get and remove the last added item to the stack
- Viewing the last added item on the stack without removing it is called “Peek” in C#. Other named may be used in other implementation such as ”Top”

Other operations

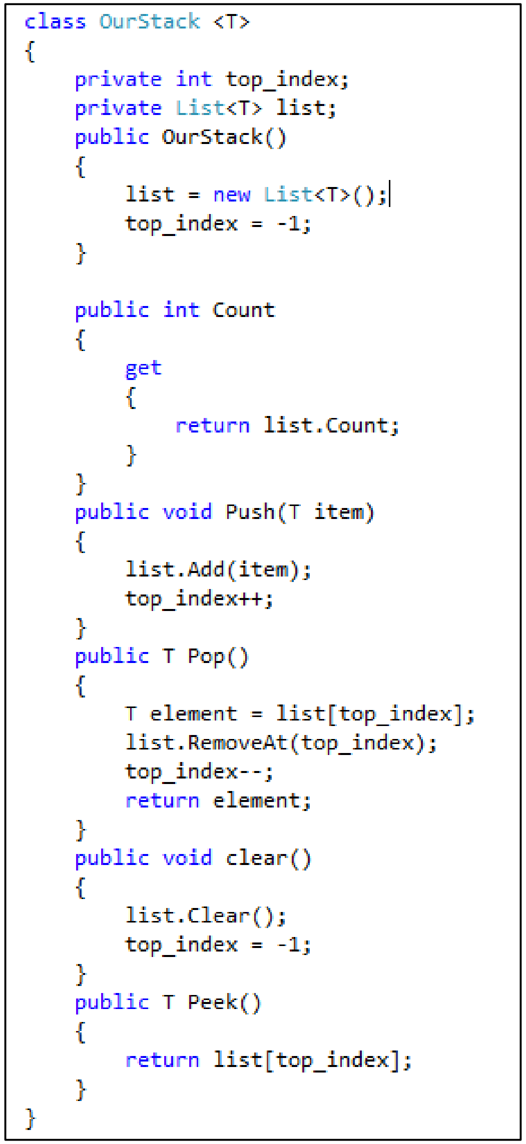
- Count: the number of elements in the stack
- Clear: Clear all the stack





Stack implementation

- What type the stack accept?
- What is the size of the stack?
- Is isFull and ISEmpty are necessary ? Why?
- What if the standard array was used instead of the List class?
- Note: The code can be enhanced to protect the Peek and POP functions from pooping or peeking from an empty stack but this will complicate the code and is left for student

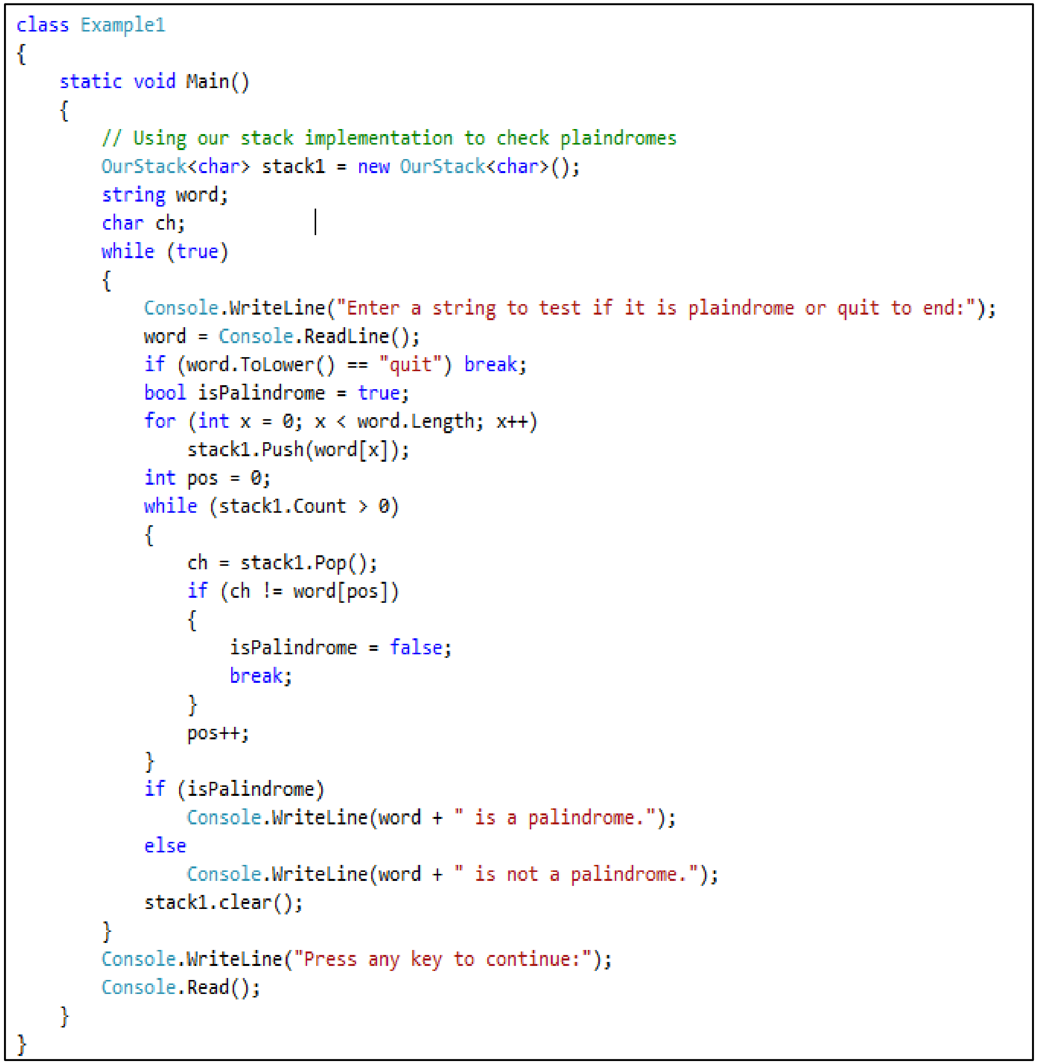


```
class OurStack <T>
{
    private int top_index;
    private List<T> list;
    public OurStack()
    {
        list = new List<T>();
        top_index = -1;
    }

    public int Count
    {
        get
        {
            return list.Count;
        }
    }
    public void Push(T item)
    {
        list.Add(item);
        top_index++;
    }
    public T Pop()
    {
        T element = list[top_index];
        list.RemoveAt(top_index);
        top_index--;
        return element;
    }
    public void clear()
    {
        list.Clear();
        top_index = -1;
    }
    public T Peek()
    {
        return list[top_index];
    }
}
```

Using stack to check palindromes

- Palindromes are reversible strings such as “”dad”, “madam”, and “sees”
- The program uses a stack to push the string character by character, then popping it character by character
- If the popped string is the same as the original staring, the string is palindrome
- Note: Braking the while loop at the first time the condition of palindrome is violated



```
class Example1
{
    static void Main()
    {
        // Using our stack implementation to check plaindromes
        OurStack<char> stack1 = new OurStack<char>();
        string word;
        char ch;
        while (true)
        {
            Console.WriteLine("Enter a string to test if it is plaindrome or quit to end:");
            word = Console.ReadLine();
            if (word.ToLower() == "quit") break;
            bool isPalindrome = true;
            for (int x = 0; x < word.Length; x++)
                stack1.Push(word[x]);
            int pos = 0;
            while (stack1.Count > 0)
            {
                ch = stack1.Pop();
                if (ch != word[pos])
                {
                    isPalindrome = false;
                    break;
                }
                pos++;
            }
            if (isPalindrome)
                Console.WriteLine(word + " is a palindrome.");
            else
                Console.WriteLine(word + " is not a palindrome.");
            stack1.clear();
        }
        Console.WriteLine("Press any key to continue:");
        Console.Read();
    }
}
```

.Net Stack implementation

```
//Stack of objecs in System.Collection
Stack myStack = new Stack();
//Generic stack in System.Collections.Generic
Stack<T> myStack = new Stack<T>();
// a stack can be created from another collection
string[] names = new string[] {"Raymond", "David", "Mike"};
Stack nameStack1 = new Stack(names);
Stack<string> nameStack2=new Stack<string>(names );
// you can specify the initial capacity of the stack when creating it
Stack myStack = new Stack(25);
// popping the stack
String onName=(string) nameStack1.Pop();
String oneName2=nameStack2.Pop();
// you can convert the stack to an array
String[] namesArray=namestack2.ToArray();
```

Report Discussion

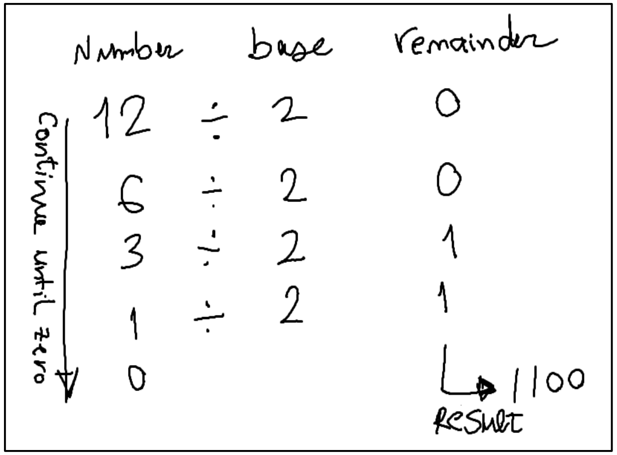
Last lecture report: Time comparison between iterative and recursive implementation of the binary search algorithm

New report:

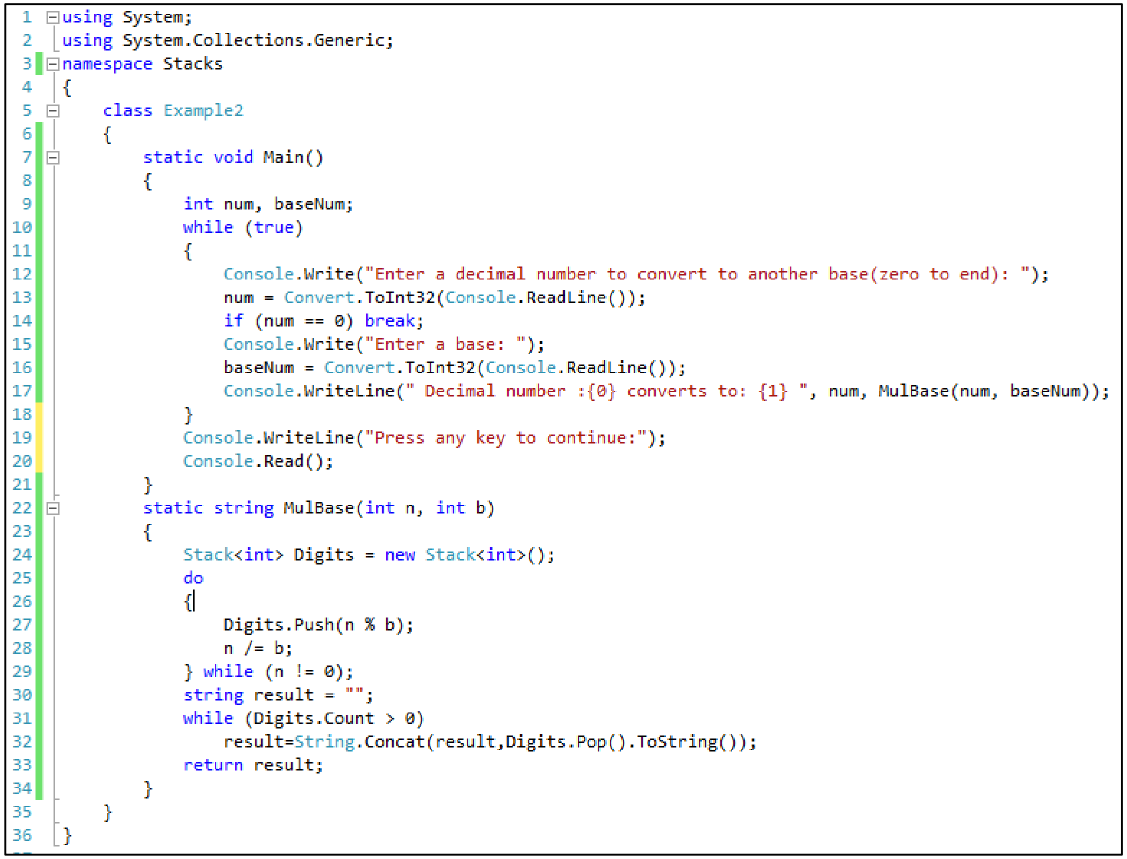
- DOTNET Stack class methods and properties
- The stack role in function calls

Nulls in C#

Stack application: Mutlti-base conversion



Number		base	remainder
12	÷	2	0
6	÷	2	0
3	÷	2	1
1	÷	2	1
0			



```
using System;
using System.Collections.Generic;
namespace Stacks
{
    class Example2
    {
        static void Main()
        {
            int num, baseNum;
            while (true)
            {
                Console.Write("Enter a decimal number to convert to another base(zero to end): ");
                num = Convert.ToInt32(Console.ReadLine());
                if (num == 0) break;
                Console.Write("Enter a base: ");
                baseNum = Convert.ToInt32(Console.ReadLine());
                Console.WriteLine(" Decimal number :{0} converts to: {1} ", num, MulBase(num, baseNum));
            }
            Console.WriteLine("Press any key to continue:");
            Console.Read();
        }
        static string MulBase(int n, int b)
        {
            Stack<int> Digits = new Stack<int>();
            do
            {
                Digits.Push(n % b);
                n /= b;
            } while (n != 0);
            string result = "";
            while (Digits.Count > 0)
                result=String.Concat(result,Digits.Pop().ToString());
            return result;
        }
    }
}
```

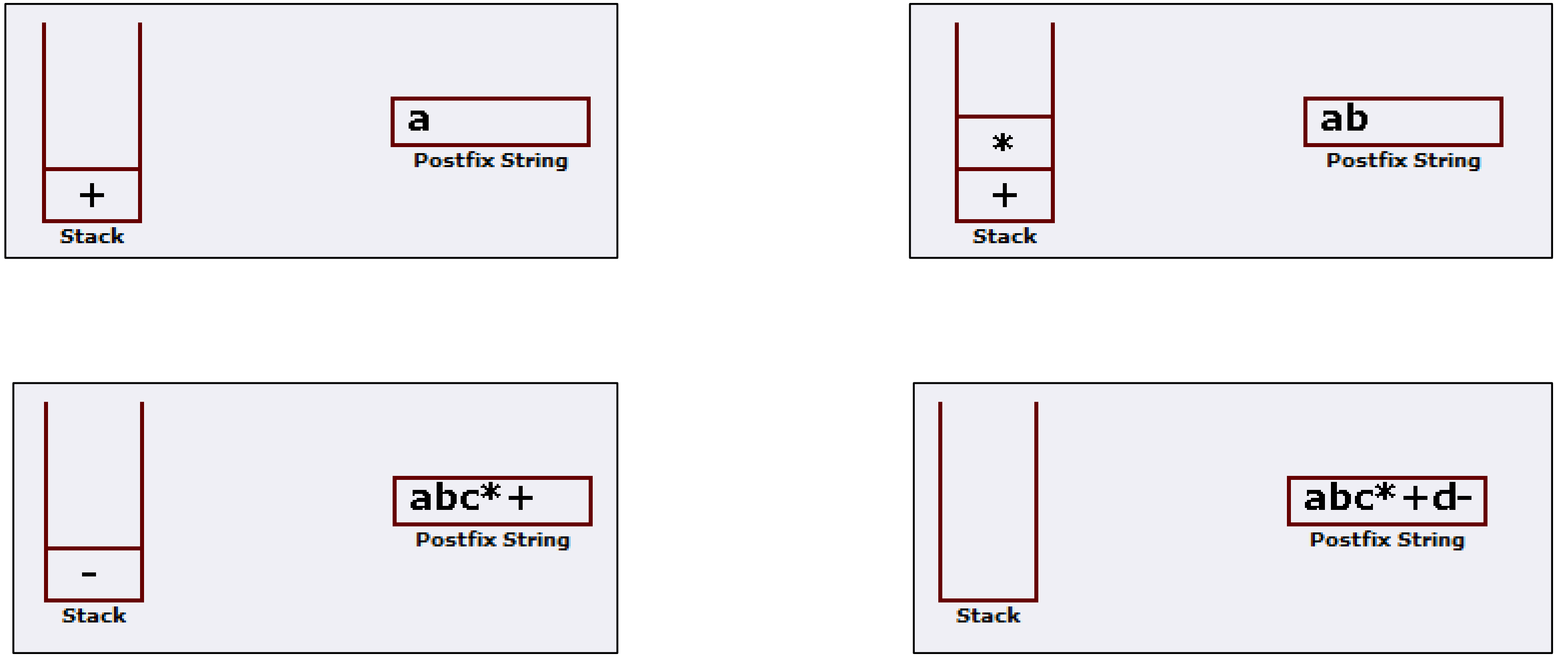
Mathematical expression: infix A+B, Prefix +AB, Postfix AB+. Postfix evaluation is more easy

Conversion from infix to postfix using a stack:

```
Define a stack for operators
Go through each item (operand/operator) in the infix expression string
If it is operand, append it to output postfix expression string.
If it is left brace push to stack
If it is operator *+-/ then
    If the stack is empty push it to the stack
    If the stack is not empty then start a loop:
           If the top of the stack has higher or equal precedence compared to its precedence
           Then pop and append to output postfix string
           Else break
        Push it to the stack
If it is right brace then
     While stack not empty and top not equal to left brace
     Pop from stack and append to output postfix string
     Finally pop out the left brace.
If there is any input in the stack pop and append to the output postfix string.
```

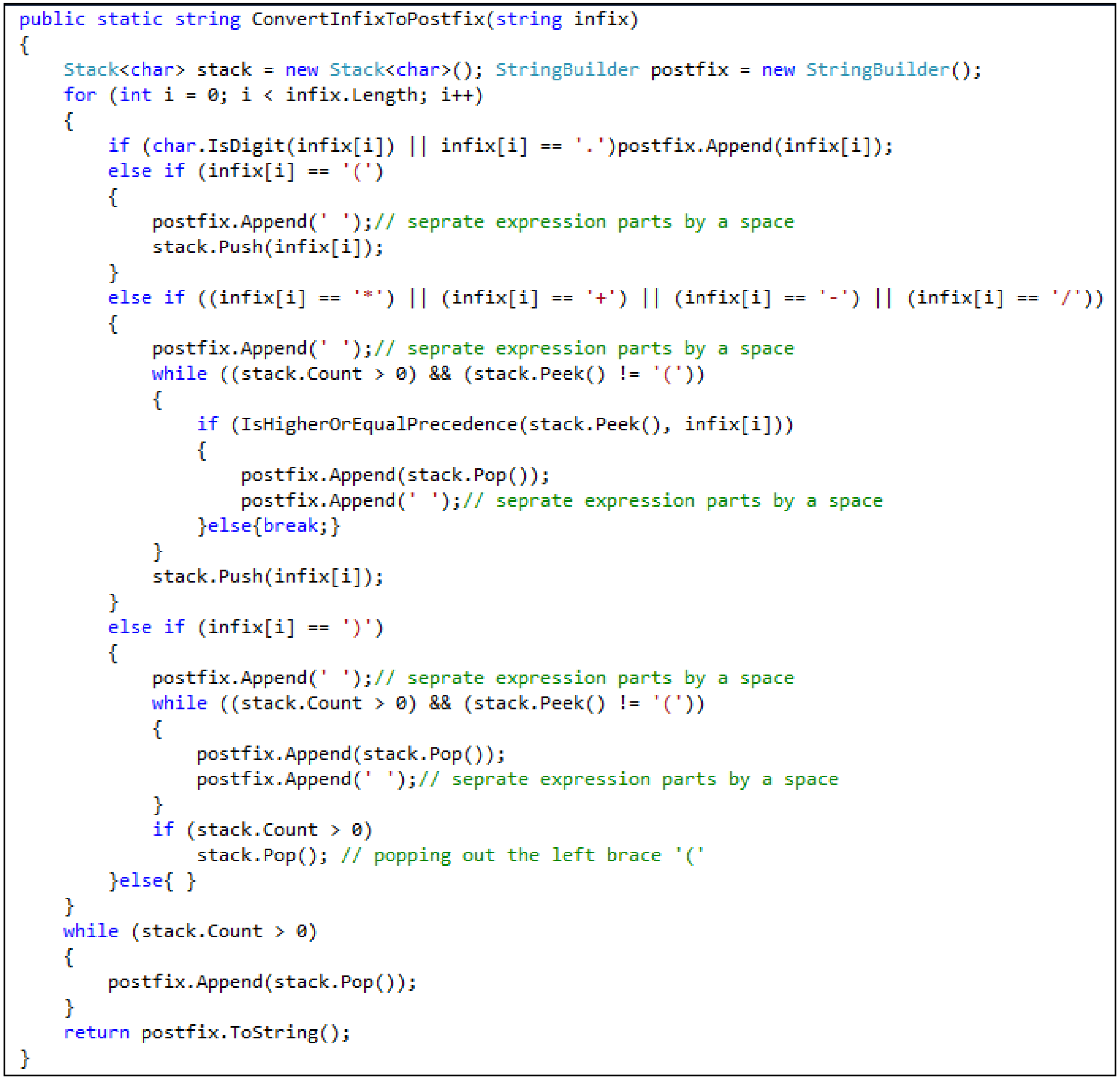
Infix to postfix conversion using a stack: Example

Infix String: $a + b * c - d$



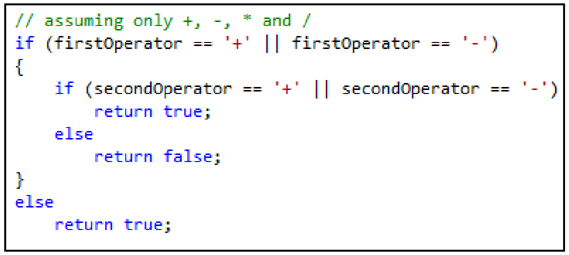
Postfix String : a b c * + d -

Conversion from infix to postfix



```
public static string ConvertInfixToPostfix(string infix)
{
    Stack<char> stack = new Stack<char>(); StringBuilder postfix = new StringBuilder();
    for (int i = 0; i < infix.Length; i++)
    {
        if (char.IsDigit(infix[i]) || infix[i] == '.')postfix.Append(infix[i]);
        else if (infix[i] == '(')
        {
            postfix.Append(' ');// seprate expression parts by a space
            stack.Push(infix[i]);
        }
        else if ((infix[i] == '*') || (infix[i] == '+') || (infix[i] == '-') || (infix[i] == '/'))
        {
            postfix.Append(' ');// seprate expression parts by a space
            while ((stack.Count > 0) && (stack.Peek() != '('))
            {
                if (IsHigherOrEqualPrecedence(stack.Peek(), infix[i]))
                {
                    postfix.Append(stack.Pop());
                    postfix.Append(' ');// seprate expression parts by a space
                }else{break;}
            }
            stack.Push(infix[i]);
        }
        else if (infix[i] == ')')
        {
            postfix.Append(' ');// seprate expression parts by a space
            while ((stack.Count > 0) && (stack.Peek() != '('))
            {
                postfix.Append(stack.Pop());
                postfix.Append(' ');// seprate expression parts by a space
            }
            if (stack.Count > 0)
                stack.Pop(); // popping out the left brace '('
        }else{ }
    }
    while (stack.Count > 0)
    {
        postfix.Append(stack.Pop());
    }
    return postfix.ToString();
}
```

IsHigherOrEqualPrecedence



```
// assuming only +, -, * and /
if (firstOperator == '+' || firstOperator == '-')
{
    if (secondOperator == '+' || secondOperator == '-')
        return true;
    else
        return false;
}
else
    return true;
```

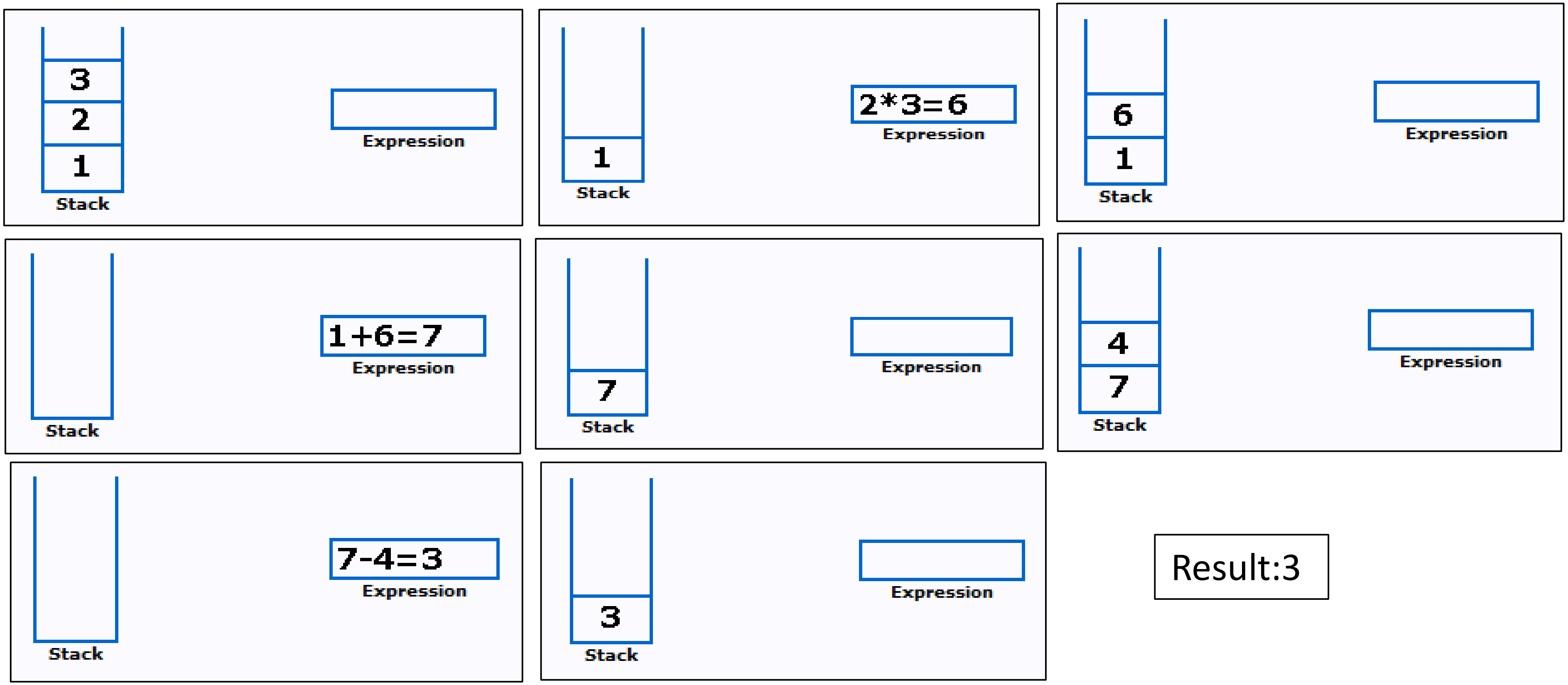
Evaluating a postfix expression using a stack

Evaluation of Postfix expression using a stack

- Initialize an empty stack for operands.
- Scan the Postfix expression string from left to right.
- If the scanned item is an operand, add it to the stack. If the scanned character is an operator, there will be at least two operands in the stack, pop and keep them, apply the operators on them(TopStack-1 operator TopStack), push the result on the stack. Repeat this step till all the characters are scanned.
- After all characters are scanned, we will have only one element in the stack which is the result of the expression.

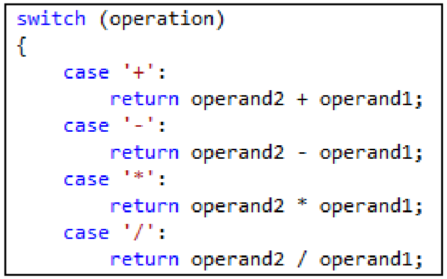
Evaluation of a postfix expression using a stack: Example

$Postfix\ expression: 1\ \ 2\ \ 3 * +4 -$

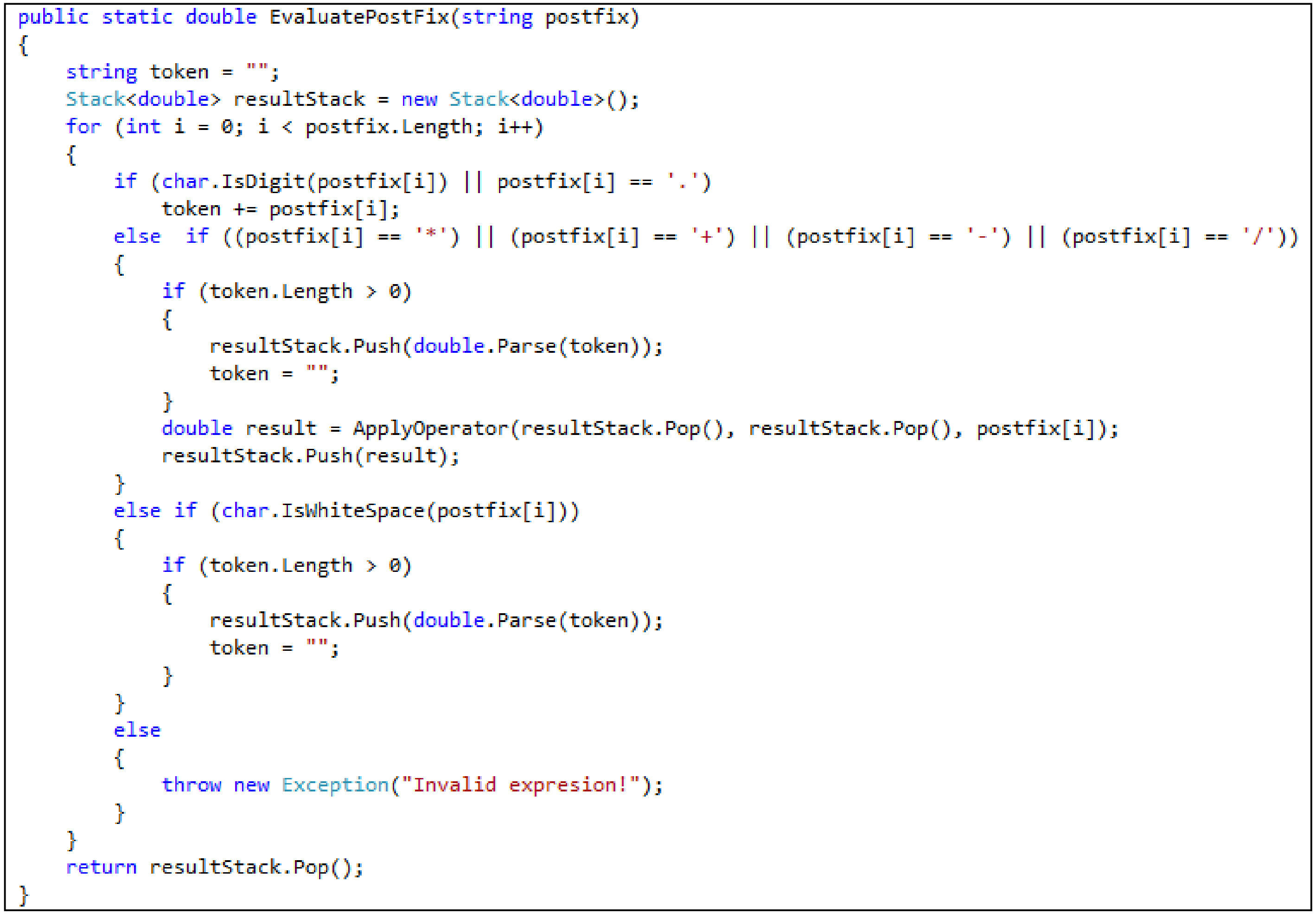


Result:3

Evaluation of postfix expression



```
switch (operation)
{
    case '+':
        return operand2 + operand1;
    case '-':
        return operand2 - operand1;
    case '*':
        return operand2 * operand1;
    case '/':
        return operand2 / operand1;
```



```
public static double EvaluatePostFix(string postfix)
{
    string token = "";
    Stack<double> resultStack = new Stack<double>();
    for (int i = 0; i < postfix.Length; i++)
    {
        if (char.IsDigit(postfix[i]) || postfix[i] == '.')
            token += postfix[i];
        else  if ((postfix[i] == '*') || (postfix[i] == '+') || (postfix[i] == '-') || (postfix[i] == '/'))
        {
            if (token.Length > 0)
            {
                resultStack.Push(double.Parse(token));
                token = "";
            }
            double result = ApplyOperator(resultStack.Pop(), resultStack.Pop(), postfix[i]);
            resultStack.Push(result);
        }
        else if (char.IsWhiteSpace(postfix[i]))
        {
            if (token.Length > 0)
            {
                resultStack.Push(double.Parse(token));
                token = "";
            }
        }
        else
        {
            throw new Exception("Invalid expresion!");
        }
    }
    return resultStack.Pop();
}
```